Direct Policy Search Method in Fault Tolerant Autonomous Systems

B. Abu Bakar * S.M. Veres **

* Faculty of Engineering, Science and Mathematics, School of Engineering Sciences, University of Southampton, SO17 1BJ, Southampton, U.K. (e-mail: bhab1g08@soton.ac.uk).
** Faculty of Engineering, Science and Mathematics, School of Engineering Sciences, University of Southampton, SO17 1BJ, Southampton, U.K. (e-mail: S.M.Veres@soton.ac.uk).

Abstract: This work is concerned with a new type of realtime reconfigurable control systems that is based on the use of a multi-agent system. To this end, two stages have been examined in the context of decision making; the fault detection and identification (FDI) stage and the reconfiguration stage (RC). The agent based FDI detects that a fault has occurred. It then further diagnoses the situation. The RC stage follows this by adapting or changing the control architecture to accommodate the fault. The agent based problem is to synchronize or integrate these two stages in the overall structure of a control system on real world applications. The multi-agent architecture proposed in this paper has several advantages in terms of modularity, reliability, ability to learn and achieve overall higher robustness over past “single software” methods. Specifically, this paper concentrates on one of the agents introduced, namely the Reconfiguration agent. Tests on a simulation of an example system have been carried out to demonstrate this new organisation of reconfigurable control systems.

Keywords: Reconfigurable control, FDI, Intelligent systems, Machine learning, Control application, Multi agent

1. INTRODUCTION

Automated controllers provide the means for manipulating a plant’s behaviour in a desired manner. A properly designed controller prevents the plant from operating in a dangerous and unstable mode due to malfunction of hardwired components. A poorly designed controller on the other hand, will result in significant damage to the plant and/or injury to people. Growing demand for safety, reliability and survivability in automated and autonomous systems have called for designing controllers that can absorb these imperfections in design and conditions. Conventional robust controller designs have become increasingly inadequate to cater for these complex systems. See Aberkane (2008). The reason behind this is that robust controllers only cater to a predefined class of faults and therefore have limited capability of reconfiguration.

The design of such controllers can be approached in many ways. Fault tolerant control systems (FTCS), as this field is termed, has been rich with research activity. In the literature, reconfigurable control systems fall under the scheme of active fault tolerant control systems (AFTCS). See Moerder (1989). It is distinguished from its counterpart, passive fault tolerant control systems (PFTCS) through the integration of both the fault detection and identification scheme (FDI) and a reconfigurable controller (RC) in the overall structure. See Zhang (2008). Most of the research done in this field have been motivated by aircraft flight control designs. Adaptive control and robust control are just two of the many approaches that have been developed. See Bodson (1997), Kwakernaak (1993), Thanapalan (2006), Wise (2006), Zames (1981). Due to the complexity of the problem, research in the past either concentrated on the fault detection and identification part (FDI) or on the control reconfiguration part (RC). Only a few research papers have been devoted to both components in the same framework.

This paper is organized as follows. The next section gives the problem description. Section 3 describes some of the related work done so far. In section 4 the architecture of the multi agent system is described. Section 5 discusses the two stages that need to be integrated. An example is given in section 6 to demonstrate the concept. This paper closes with section 7 which concludes the work and details future research to be done.

2. PROBLEM DESCRIPTION

The problem our work investigates can be summarized as follows. An online reconfigurable controller is desired due to (1) malfunction of actuator components (2) partial malfunction of sensor components. To this end, two stages have to be looked at; the fault detection and identification (FDI) stage and the reconfiguration stage (RC). The FDI stage detects that a fault has occurred. It further diagnoses the situation. The RC stage on the other hand, adapts or changes the control architecture to accommodate the fault. The problem is to synchronize or integrate these two stages in the overall structure of a control system in real world applications. Efforts to combine these two stages have been made in the past as described in section 3. However, an effective integration in practice still poses a challenging problem. These problems include deciding when is the right time to reconfigure; the accuracy of the FDI component in providing information to the RC component; modelling mismatch between the two compo-
A combination of robust control theory and reinforcement learning to develop a stable neuro-control scheme is presented in Kretchmar (2000). The method of representation for the nonlinear and time varying components of the neural network was to use functional uncertainty. Robust control techniques were applied to guarantee the stability of the neuro-controller. The scheme provides stable control not only for a specific fixed-weight, neural network, but also for a neuro-controller in which weights change during learning. It uses the actor-critic reinforcement learning algorithm as a controller. The reinforcement learning algorithm was chosen because it is well suited to the type of information available in the control environment. Research related to controllers using reinforcement learning was also found in Ng (2000), Van-Hasselt (2007). In the former, a direct policy search algorithm with dynamic programming was used to control a helicopter. The algorithm assumes that a model is present. In the latter, a new class of algorithm named Continuous Actor Critic Learning Automaton (CACLA) was introduced. It takes advantage of the temporal learning method which does not require a model of the environment.

A more direct approach to reconfigurable control is to use an adaptive systems architecture. In this approach a diagnosis module is omitted and faults are not detected through explicit FDI. It has been quite successfully implemented in aircraft and weapons systems. Work done in this field can be found in Bodson (1997), Thanapanal (2006).

A relatively new approach to computing is the use of a multi agent architecture in complex software systems. Useful control techniques provided by multi-agent systems for modular, metamorphic robots were demonstrated in Bojinov (2002). An automatic reconfiguration of a robotic arm using multi agents was presented in Britain (2008). A three layer agent architecture was used to distribute tasks among agents. A reconfiguration is carried out every time a request is made by the robot. It stops short of describing the actual mechanism of how the request is made. It also does not explain how a behavioural pattern is chosen. All the work above assumed that all the sensors and actuators were working in their nominal mode and did not deal with fault analysis.

A multi agent approach for control systems was proposed in Veres (2004). This architecture contains agents for various components of the system, for example modelling and controller optimization. This new breed of agents are called cautiously optimistic control agents or COCA which applies new modelling results with caution while still using current settings until a certain threshold is exceeded. The key feature is that the central unit does not have full authority over the agents’ response. The cooperative action between the different components is what makes the architecture successful.

This work extends the concept in Veres (2004) to be used for FDI & RC integration. A multi agent architecture will be used to interface between the two stages. As described in the introduction, machine learning algorithms will be used to equip the agents with a high level of intelligence. It allows the system to dynamically evolve. Not only will it be able to learn unmodelled faults, it will also learn new solutions to deal with the faults. In relation to Bojinov (2002), instead of having multiple agents in one layer, this work will implement a multiple layer structure. Each agent in a different layer has a specific task. In contrast to Britain (2008), this work will incorporate fault diagnosis in the overall system. It also extends...
the intelligence of some agents beyond simple rules and logic as described in Bojinov (2002) and Britain (2008). With regards to Ng (2000) and Van-Hasselt (2007), our work combines the two approaches to develop a new method for choosing optimal actions at any given state.

4. ARCHITECTURE

The multi agent system for fault tolerant control will be defined similarly to the ones proposed in Veres (2004). It will have the following agents:

(1) Planner agent \(A_p\)
(2) Monitoring agent \(A_m\)
(3) Reconfiguration agent \(A_r\)

In general a physical agent \(A\) will be a tuple defined by

\[ A = \langle \Pi, \Sigma, \Omega \rangle \]  

where \(\Pi\) is its physical engine, \(\Sigma\) is its rational behaviour engine and \(\Omega\) is its continuous engine. Not all three components will be active in particular agents and some may only contain \(\Pi\) or \(\Sigma\). If present, \(\Omega\) is a computational unit to compute the future evolution of continuous time models in the environment or in the agents body. \(\Sigma\) is a temporal logic processor that takes assumptions and goals and computes plans to achieve goals under constraints. \(\Pi\) contains access to communication devices and connections to sensors and actuators and can also contain a set of parametrised feedback or feedforward and open loop controllers that enables the agent to interact with its environment via the use of specific groups of the agent’s sensors and actuators. This representation method of a basic physical agent serves important practical purposes. It allows for easy programming by engineers, formal verifiability of the agent’s body and access to communication devices and connections to sensors and actuators and can also contain a set of parametrised feedback or feedforward and open loop controllers that enables the agent to interact with its environment via the use of specific groups of the agent’s sensors and actuators. This representation method of a basic physical agent serves important practical purposes. It allows for easy programming by engineers, formal verifiability of the system and fast realtime operation on computers. In general, the physical engine \(\Pi\) is a quad tuple defined by

\[ \Pi = \langle \gamma, \sigma, \alpha, \Delta \rangle \]

where \(\gamma\) is a set of communication devices, \(\sigma\) is a set of sensor data abstractors, \(\alpha\) is a set of control data developers and \(\Delta\) is a set of realtime symbolic feedback loop executors. Any of the \(\Delta, \alpha, \sigma\) can be empty. Since \(\gamma\) represents communication devices, it can never be empty. An agent that only has \(\gamma\) defined is called a software agent.

The rational behaviour engine \(\Sigma\) is a tuple defined by

\[ \Sigma = \langle W, M_p, M_f, G, R \rangle \]

where \(W\) is a world model, \(M_p\) is an abstract physical skills memory, \(M_f\) is a solution memory, \(C\) is an abstract formulation of behaviour constraints, \(G\) is an abstract formulation of short and long term goals and \(R\) is a reasoning cycle controller.

This general implementation platform can fit popular agent architectures well. In the case of Belief Desire Intention (BDI) agents, \(W\) can represent the beliefs of an agent, \(M_p\) can mean the plans of an agent, \(M_f\) can be associated with relative frequencies (probabilities) of plans succeeding in the past, \(G\) could be the goals that it wants to achieve.

The continuous engine \(\Omega\) is a tuple defined by

\[ \Omega = \langle M, S, O, B, L \rangle \]

where \(M\) is a set of approximate continuous models of the world, \(S\) is a continuous time simulator that uses analytical and empirical data based dynamical models to predict the future state of the world, \(O\) is an optimizer, \(B\) is a Boolean evaluator of propositions in terms of \(\sigma\) statements and \(L\) is a library of knowledge.

The individual agents can then be defined as follows. \(A_p\) is of the form \(\langle \Sigma_p, \Omega_p \rangle\), \(A_m\) is of the form \(\langle \Pi_m, \Sigma_m, \Omega_m \rangle\) whereas \(A_r\) is of the form \(\langle \Pi_r, \Sigma_r, \Omega_r \rangle\). The agents will be implemented in a multilayer structure. At the top, \(A_p\) will be supervising the overall progress of planning and tasks. This configuration allows the designer to include prior knowledge of system functionalities. A checking of the system status is done via \(A_m\) which enables unmodelled faults to be taken into consideration. The reconfiguration of the system is done via \(A_r\).

5. FDI AND RECONFIGURATION

The agents need a communication systems to execute fault diagnostics and reconfiguration. Agent tasks are initiated by messages received from other agents. The nature of communications are depicted in Figures 3-5 for three scenarios to show how the agents interact.

In figure 3, the default scenario is shown when now fault is detected by the monitoring agent. The flow of events are numbered according to order of occurrence. The planner agent

Fig. 2. Multi-agent structure

Fig. 3. Flow diagram of events - Default
A_p holds a set of predefined default solutions in \( M_p \) of its rational behaviour engine \( \Sigma_p \). It chooses a default solution of control and sends the parameters to the reconfiguration agent \( A_r \). These parameters and constraints are loaded into the rational behaviour engine \( \Sigma_r \) of the reconfiguration agent \( A_r \) and executed by \( \Pi_m \). In case of no fault the operational mode of \( A_r \) is set to 'not learning' since we assume that in the default mode there are no faults and the system has been properly trained in terms of control methods prior to implementation.

The physical engine output of \( A_r \) is the action taken by the AGV in terms of motion primitives. The monitoring agent \( A_m \) assesses the state of the AGV using \( \Pi_m \) and \( \Omega_m \) if the result of an action is what was expected. In the event that there are no faults, \( A_m \) sends a 'no faults' flag and \( A_p \) stays with the control solution previously chosen.

In Figure 4, the fault detection scenario is shown. Here we assume that the system is already executing a successful solution. The flow of events is shown numbered from 1-6 in order of occurrence. When the monitoring agent \( A_m \) assesses a state and detects a fault, it sends a 'fault' flag and passes the diagnostics results to \( A_p \). Upon receiving this from \( A_m \), the planner agent \( A_p \) uses its \( \Omega_p \) to decide the conditions of which predefined solution best match the fault characteristics. The best solution is then sent to \( A_r \) to be executed. \( A_r \) then changes its mode to 'learning' and disables the monitoring agent so that no further fault flag is raised while \( A_r \) is finding/learning a solution. It is important to note that an a priori solution need not be perfect. It is up to \( A_r \) to improve on the solution. The \( A_r \) is provided with a priori knowledge so that it does not have to learn from scratch.

Fig. 5 shows how a reconfiguration scenario executed. The sequence of events is again numbered in the figure. When a fault is detected and \( A_r \) has decided on a solution to try, \( A_r \) loads the parameters into its memory and starts the learning process. It tries to learn with the behaviour constraints defined in \( C \) of its \( \Sigma_r \). The goal is to find a set of actions or policy that will minimize the error caused by the fault. \( A_r \) repeats this process until it has converged to a good solution. Once it has found a solution, the parameters are sent to \( A_p \) to be stored in \( M_p \) of \( \Sigma_r \). The system has reconfigured successfully and the monitoring agent is again enabled so that fault diagnostics can be resumed.

The most important aspect of this scheme is that the agents are required to have a degree of intelligence to make decisions. In this the agent’s continuous engine \( \Omega \) plays a vital role.

Algorithms that are used in each agent’s continuous engine are described as follows.

We will first look at \( \Omega_r \). One of the vital characteristics of a reconfiguration agent \( A_r \) is that it must be able to find new solutions. The only practical way of doing so is by including the use of “trial and error” in a reinforcement learning algorithm. In reinforcement learning, the idea is to try and maximize the amount of reward a learner gets in achieving a goal state from a starting state. It does so by interacting with its environment and taking actions based on these interactions. A reinforcement learning agent will choose previous actions found to be effective in producing large rewards. However, such actions first need to be discovered. Only when actions that it has not selected before is tried can the agent know what rewards it will receive. It has to exploit its knowledge in order to obtain reward, but it also has to explore in order to make better action selections in the future. See Sutton (1998). This characteristic fits in naturally with the learning needs of the reconfiguration agent \( A_r \) to whom not all solutions need not be known a priori.

A reinforcement learning task can be seen as a Markov decision process (MDP). A system is said to have the Markov property if all the information needed to predict any future state \( S_{n+1} \) only depends on the current state and action pair \( S_n, a_n \). Figure 6 shows how states and state action pairs in a reinforcement learning task are represented as an MDP. State nodes are represented by a large open circle with the label \( S_n \). Action nodes for each state-action pair are represented by a small solid circle with the labels \( S_n, a_n \). The reward for transitioning to state \( S_{n+1} \) by taking an action \( a_n \) in the state \( S_n \) is denoted as \( r_{n+1} \). So a reward of \( r_1 \) will result when transitioning to state \( S_1 \) by taking the action \( a_0 \) in the state \( S_0 \).

This scheme works fine for finite set of states. For practical applications however, the number of states is infinite. Fortunately, the states can be generalized using linear function approximation. In our example, a variety of the direct policy search method was implemented. See Baxter (2000), Williams...
A new method for selecting actions from a continuous action set is introduced.

The direct policy search algorithm maps the inputs directly to actions. It does not require an action-value function to be calculated. In general, the goal is still to find a policy \( \pi : S \times A \rightarrow \mathbb{R} \) that maximizes its expected reward (Williams (1992)). The algorithm optimizes the policy by executing gradient ascent updates on the policy function.

On the other hand, the monitoring agent needs to assess whether a fault has occurred. Its continuous engine \( \Omega_p \) has to have the means to differentiate between a normal and a faulty condition. Here, implementation of a classification algorithm is a viable means to differentiate between a normal and a faulty condition. The agent could be trained to look for patterns of faults from features of the input. A concrete example of how this is done is discussed in Section 6.

A planner agent’s continuous engine \( \Omega_p \) needs to have the capability of searching through its solution memory \( M_g \) of \( \Omega_p \) and deciding on the best course of action. This is implemented through simple logic statements that can include complex propositions such as applying algorithms to transfer experience across different scenarios.

6. IMPLEMENTATION

A sample system in the form of an autonomous ground vehicle (AGV) model was used to demonstrate the concept of multi-agent architecture.

For the reconfiguration agent, a variety of the direct policy search reinforcement learning was implemented as described in the previous section. In our work, a slight modification to the update rule is made which results in a simpler implementation. In previous algorithms, the update is dependent on the value of the reward. Here, only the sign of the change of rewards is taken into account.

Specifically, let \( \pi_t(s, a) \) be the probability of selecting action \( a \) in state \( s \) at time \( t \). Since our example system has four actuators, the action is actually a vector with four elements. Here, the policy is defined as in Van-Hasselt (2007) to be

\[
\pi_t(s, a) = \frac{1}{2\pi\sigma} \exp\left(-\frac{(a-a'(s))^2}{2\sigma^2}\right)
\]

where the action \( a \) is the output chosen by sampling around the mean \( a'(s) \) and standard deviation \( \sigma \) at time \( t \). \( a'(s) \) is the current approximate optimal value given by a function approximator. It is also worth noting that \( \pi_t(s, a) \) is the policy and \( \pi \) is a mathematical constant.

A 3 layer back propagation neural network with 5 inputs, 2 hidden nodes and 1 output for each of our four independent motors in our example system was chosen to approximate \( a'(s) \). Each output is defined as \( a' \in [-1,1] \). The five inputs are listed in table 1.

Table 1. Inputs of network

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v )</td>
<td>Velocity of AGV</td>
</tr>
<tr>
<td>( \phi )</td>
<td>Heading angle</td>
</tr>
<tr>
<td>( e_v )</td>
<td>Error between reference velocity and actual velocity</td>
</tr>
<tr>
<td>( e_{\phi} )</td>
<td>Error between reference heading angle and actual heading angle</td>
</tr>
<tr>
<td>( \omega )</td>
<td>Angular velocity of AGV</td>
</tr>
</tbody>
</table>

Fig. 7. Neural network for \( a'(s_t) \)

where \( \delta_t = r_{t+1} - r_t \) is the change of reward from time \( t \) to time \( t+1 \), \( \theta_{t+1}^{a'} \) is the parameter or weights of the function \( a' \) and \( \alpha \) is the learning rate. The reward function is defined as

\[
r_t = -e_v^2 - e_{\phi}^2
\]

The reward function given above encourages the learner to minimize the error of the states faster in the shortest amount of time. Intuitively Eq. 6 means that if an action sampled from Eq. 5 leads to a immediate reward higher than the previous step, then we reinforce the action by edging \( a'_t(s_t) \) to output something closer to \( a_t \). This update rule is different from Van-Hasselt (2007) in which the actor-critic method was used and the update was done corresponding to the change in value function. Following Sutton (1998), the update function given in Eq.6 is guaranteed to converge to a local optimum under stochastic approximations for a decreasing learning rate \( \alpha \).

In our example, the goal is to have the error between reference states and actual states to be zero. The outputs \( a' \) corresponds to the torque applied to each motor \( \tau_{LF}, \tau_{LB}, \tau_{RF}, \tau_{RB} \). The learner is given 3000 time steps to reach the goal before the episode is terminated and a new episode is started. In our example, “time step” means one simulation cycle and “episode” means a single trial in which the learner takes many time steps to reach its goal. This could result in the learner actually reaching its goal or it could exceed the number of time steps allowed. The tracking problem can be thought of as a finite state MDP. Once the goal state is achieved or the maximum number of steps are taken, the episode ends. A new episode is started and the learning begins again with an improved set policy. In order to make sure that proper exploration and exploitation of possible actions takes place, the standard deviation for the gaussian exploration policy defined previously is given an initial value of 0.7. The standard deviation is then decreased by multiplying Eq.8.
\[ \epsilon_{ep+1} = \epsilon_{ep} \frac{1}{\epsilon_{ep}^{0.2}} \]  

(8)

where \( \epsilon_{ep} \) is the exploration rate and \( ep \) is the current episode of learning. This ensures that as the agent’s experience grows, the optimal action is taken. In the initial learning stage, a different reference velocity and reference heading angle is generated for each episode. The initial learning process is repeated until the agent converges to an optimal policy \( \pi^*(s, a) \). The model used to train the algorithm is given in Caracciolo (1999) as

\[ M\dot{\eta} + C\dot{\eta} + R = B\tau \]  

(9)

where

\[
\eta = \begin{bmatrix} v_x \\ \omega \end{bmatrix} \\
\tau = \begin{bmatrix} \tau_L \\ \tau_R \end{bmatrix} \quad \text{where} \quad \tau = \begin{bmatrix} \tau_L F - \tau_R F \\ \tau_R F - \tau_L F \end{bmatrix} \\
C = \begin{bmatrix} mx_{ICR} & 0 \\ 0 & -\dot{\phi} \dot{x}_{ICR} \end{bmatrix} \\
M = \begin{bmatrix} m & 0 \\ 0 & mx_{ICR}^2 + I \end{bmatrix} \\
R = \begin{bmatrix} F_{ix}(q) \\ x_{ICR}F_{iy}(q) + M_x \end{bmatrix} \\
B = \frac{1}{r} \begin{bmatrix} 1 & 0 \\ -c & 1 \end{bmatrix}
\]  

\( \eta \) has AGV velocity \( v_x \) and angular velocity \( \omega \) as components. \( \tau \) is the input torque with components \( \tau_L \) and \( \tau_R \) as the torques produced by the left and right wheels respectively, \( m \) is the mass of the AGV, \( x_{ICR} \) is the horizontal coordinate of the instantaneous center of rotation (ICR), \( \dot{\phi} \) equals the angular velocity \( \omega \) and \( I \) is the moment of inertia. \( F_{ix}(q) \), \( F_{iy}(q) \) are the generalized resistive forces in the x and y coordinate and \( M_x \) is the resistant moment around the center of mass. \( r \) is the radius of the wheel and \( c \) is half the distance between the left and right wheels. \( \dot{q} \) is the generalized velocity defined by

\[ \dot{q} = S(q)\dot{\eta} = \begin{bmatrix} \dot{X} \\ \dot{Y} \end{bmatrix} \]  

(16)

where

\[
S(q) = \begin{bmatrix} \cos \varphi & x_{ICR} \sin \varphi \\ \sin \varphi & -x_{ICR} \cos \varphi \end{bmatrix}
\]  

(17)

\( X \) and \( Y \) are the x-axis and y-axis velocities respectively, \( \dot{\varphi} \) is once again the angular velocity. For a detailed explanation to how the equations were derived, interested readers are referred to Caracciolo (1999). In our example, the values of \( v_x \) and \( \varphi \) are desired. \( \varphi \) is obtained by integrating equation 16.

The weights of the function approximator were initialized to random values. A high number of episodes were run during the training phase. For each episode, a new reference value is given randomly so that as many states as possible are visited to ensure proper convergence. After the agent’s policy has converged, the algorithm is tested on the AGV model with the optimal policy \( \pi^*(s, a) \).

Fig. 8 shows the mean squared error of the output from the function approximation. From the figure it can be seen that the mean squared error decreases and converges to about 0.05 as the episode increases.

Fig. 9 and Fig. 10 depicts how well the AGV tracks a reference value. The red line is the reference value whereas the blue line is the actual value of the AGV. In Fig. 9, the reference velocity changed every episode while keeping the reference heading angle constant. In Fig. 9, the reference heading angle is changed each episode while the reference velocity is kept constant. The two figures show that the AGV is able to track the reference values up to an acceptable error.

A monitoring agent checks to see whether the goals have to be reassessed. It does so by classifying healthy and faulty states. This is done by teaching the monitoring agent to look for patterns of different states. The data can be modelled as a mixture of gaussians. The EM (Expectation-Maximization) algorithm can then be used to fit the parameters of the model. The EM algorithm is an iterative two step algorithm. In the E-step, it makes a guess on the values of the classes. In the M-step, it updates the parameters of the model using the guess made in the E-step. It does this continuously until convergence.
calculate the parameters to pzing algorithm. The class label These algorithms fall under the category of unsupervised learning algorithm, except it assigns probability to the different classes. This is similar to the K-means class distribution respectively. In the E-step, the posterior probability of the classes given the observed data is calculated by Bayes theorem as follows

\[ w = p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma) = \frac{p(x^{(i)}|z^{(i)} = j; \mu, \Sigma)p(z^{(i)} = j; \phi)}{\sum_{l=1}^{k} p(x^{(i)}|z^{(i)} = l; \mu, \Sigma)p(z^{(i)} = l; \phi)} \]  

(18)

where \( x^{(i)} \) is the input data, \( z^{(i)} \) is the class label. \( z^{(i)} \sim \text{Multinomial}(\phi) \) (where \( \phi_j \geq 0, \sum_{j=1}^{k} \phi_j = 1 \)). Here, \( k \) is the number of values \( z^{(i)} \) can take up.

In the M-step, the parameters are updated as follows

\[ \phi_j = \frac{1}{m} \sum_{i=1}^{m} w_j^{(i)} \]
\[ \mu_j = \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}} \]  
\[ \Sigma_j = \frac{\sum_{i=1}^{m} w_j^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} w_j^{(i)}} \]  

(19)

where the parameter \( \phi_j \) gives \( p(z^{(i)} = j) \). \( m \) is the total number of inputs, \( \mu_j \) and \( \Sigma_j \) are the mean and covariance of each class distribution respectively. This is similar to the K-means algorithm, except it assigns probability to the different classes.

These algorithms fall under the category of unsupervised learning algorithm. The class label \( z^{(i)} \) is not known. Therefore, in the E-step the algorithm initiates by randomly assigning a value to \( p(z^{(i)} = j) \). In the M-step, the algorithm uses the values to calculate the parameters \( \phi_j, \mu_j \) and \( \Sigma_j \). Then, in the next E-step \( p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma) \) is recalculated using the parameter values found in the M-step. This is done iteratively until the algorithm converges to a solution. After the algorithm has been trained, class labels from our training set are compared with the class labels predicted by our model to check that the model is correct.

The learning algorithm was trained to recognize 3 states:

- Healthy
- Sensor fault
- Actuator fault

Fig. 11 shows a typical test set being classified by the algorithm. The test data is ordered such that each class corresponds to the front, middle and the end of the graph respectively. It can be seen that most of the points are correctly classified.

In the test phase, the system is run for 50 episodes at optimal configuration with all four motors running. An actuator fault is simulated by turning one of the actuators at episode 51. Fig. 12 shows how the standard deviation of the time taken to finish the last 20 episodes changes when an actuator is turned off. The top part of the Fig. 12 is the average timesteps of time taken to finish the last 20 episodes. The bottom part is the standard deviation. It rises dramatically after episode 51. When it reaches a threshold level (a threshold of 15 was determined heuristically), the monitoring agent \( A_{sn} \) recognizes that an actuator fault has occurred and passes this information to the reconfiguration agent \( A_r \). The planner agent changes strategy to operate only with two actuators. The new parameters are distributed to the reconfiguration agent \( A_r \). From the graph it can be seen that the standard deviation falls below the threshold level after only a few episodes. The system has successfully reconfigured. From the top graph of Fig. 12, it can also be seen that the system is able to reach its goals even though an actuator fault has occurred. However, it takes more time to do so. This shows that the system is robust enough to allow time for fault recognition.

7. CONCLUSIONS AND FURTHER WORKS

A multi-agent architecture to solve the integration of FDI and RC was introduced. Each agent in the architecture was defined and the components belonging the agents were described.
An example was given in sections 6 to demonstrate how the agent architecture could be defined on practical systems. It was also demonstrated how the agents behave. In the case of reconfiguration agents $A_r$, it tries to adapt to the environment by constantly learning an optimal policy. A new variation of the direct policy search algorithm was introduced. In the case where the goal cannot be achieved, the monitoring agent detects the fault and goal reassessment is carried out.

Future work will concentrate on the planner and monitoring agents. Methods such as symbolic reasoning could provide further improvements. Implementing the agents in the hardware in Fig. 1 and the the previous sections will also be one of the immediate tasks.

REFERENCES


N. E. Wu. Robust feedback design with optimized diagnostic performance. IEEE transactions on automatic control, Vol. 42(no.9), 1997.


